In: *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS '94)*
Austin, Texas, September 1994, IEEE, pages 131–139.

Divergence Caching in Client-Server Architectures*

Yixiu Huang Robert H. Sloan Ouri Wolfson
Electrical Engineering and Computer Science Dept.
University of Illinois at Chicago
Chicago, IL 60607

Abstract

In this paper we propose a new mechanism, divergence caching, for reducing access and communication charges in accessing on-line database servers. The objective is achieved by allowing tolerant read requests, namely requests that can be satisfied by out-of-date data. We propose two algorithms based on divergence caching, Static and Dynamic. The first is appropriate when the access pattern to an object in the database is fixed and known, and the latter is appropriate in other cases. We analyze these algorithms in the worst case and the expected case.

1 Introduction and overview

Users will soon have on-line access to a large number of databases, often via wireless networks. The potential market for this activity is estimated to be billions of dollars annually, in access and communication charges. For example, passengers will access airline and other carriers' schedules, and weather information. Investors will access prices of financial instruments, salespeople will access inventory data, callers will access location-dependent data (e.g. where is the nearest taxicab, see [3, 9]) and route-planning computers in cars will access traffic information.

Because of limited bandwidth, wireless communication is more expensive than wire communication. For example, a cellular telephone call costs about \$0.35 per minute, and RAM Mobile Data Corporation charges on average \$0.08 per data message to or from a mobile computer (the actual charge depends on the length of the message).

Additionally, database publishers will also charge an access fee for each transmission of an object or data item to a client. Thus each such transmission will incur both access and communication charges. Similarly, today when calling a 900 telephone number the caller is charged two separate fees, one for communication and the other for access.

It is clear that for users who perform hundreds of accesses each day, access and communication charges can become very expensive. Therefore, it is important that client computers access on-line database servers in a way that minimizes these charges.

In this paper, we explore the minimization of these charges via a new mechanism called *divergence caching*. Its objective is to reduce the number of transmissions of an object from an on-line database to a client computer. It does so by using the following two techniques for each object in the on-line database.

- The first technique is using *tolerant reads*. A Client Computer (CC) issues reads (and possibly writes) for a data object. In order to reduce access and communication charges, each read is associated with a natural number representing the divergence-tolerance for the read. For example, read(IBM,3) represents a request to read IBM's stock price (i.e. the object) with a tolerance of 3. This read can be satisfied by any of the three latest versions of IBM's stock price; in other words, it can be satisfied by a version that is up to two updates behind the most recent version.

- The second technique is *automatic refresh*. Its purpose is to eliminate the need for object transmission for every read, and it does so as follows. The Server Computer (SC) that stores the on-line database receives all the updates of the object. We are not concerned with the source of these updates. For every client computer that reads the object, the SC has a refresh rate r. This means that the version of the object cached at the CC is at most $r-1$ updates behind the version at the SC. To achieve this, the SC automatically propagates to the CC the r'th version since the last transmission of the object to the CC. The CC saves the last version of the object it received from the SC. Thus, those reads at the client computer with a tolerance greater than r can be satisfied locally; i.e., without access to the on-line database (which avoids the access and communication charges). Therefore, access and communication charges are incurred only for automatic refresh, and for each read with a tolerance that is lower than r.

*This research was supported in part by NSF grant IRI-9224605 and AFOSR grant F49620-93-1-0059.

The refresh rate can have any value between 1 and infinity. A refresh rate of 1 means that the CC has a regular copy of the object (i.e. the object is replicated at the CC in the traditional sense), and each update of the object is propagated to the CC. A refresh rate of infinity means that the CC does not have a copy of the object; each read, regardless of its tolerance, will require a transmission from the SC to the CC (even when the object has not changed since the last read). The optimal refresh rate, i.e. the refresh rate that minimizes the object transmissions, depends on the ratio between the frequency of updates at the SC on one hand, and the frequency and tolerance of reads at the CC on the other hand.

Note that this paradigm is only appropriate for cases where the client does not need the most recent value of the object. We envision these methods being used, for instance, by a relatively passive investor to monitor her portfolio, or by a basketball fan (perhaps at work) to get scores during an ongoing game, or by a customer monitoring the general level of an inventory item (e.g. the number of compact cars available at a particular rental location, with each update representing a rental or a return).

In this paper we propose and analyze two related algorithms, Static Divergence Caching (SDC) and Dynamic Divergence Caching (DDC). The SDC algorithm works as described above, and it has a fixed refresh rate. Observe that actually there is an infinite number of SDC algorithms, one for each refresh rate. One of the problems that we solve in the paper is to determine the optimal refresh rate for a given frequency of writes and reads of each tolerance. Specifically, we assume a Poisson distribution for the writes, a Poisson distribution for reads of each tolerance, and we assume that we know the intensities (i.e. λ parameters) of each type of request; and we solve the problem of determining the refresh rate that will minimize the expected cost of a request.

The DDC algorithm is similar to the static one, except that the refresh rate varies over time. It does so since we assume that the intensities of the Poisson distributions are unknown or they vary over time. Our DDC algorithm learns these by "watching" a sliding window of read-write requests, and based on it the algorithm continuously adapts the refresh rate to the current intensities. The DDC algorithm also has an infinite number of variants, one for each window-size. The DDC algorithm is distributed, and it varies the refresh rate between 0 and infinity. It is implemented by software residing on both the client and the server computers.

In addition to the expected case, we also analyze the worst case for both algorithms, and we show that in the worst case the DDC algorithm is superior to the SDC algorithm.

Finally, we analyzed the DDC algorithm experimentally. We have determined that the appropriate window size is approximately 23. We have also determined that if the distribution parameters are fixed, then the DDC algorithm comes within 15–45% of the optimal SDC algorithm, i.e. the static algorithm with the optimal refresh rate for the given distribution parameters. For fixed distribution parameters the DDC algorithm is used when these parameters are unknown a priori. If the distribution parameters vary over time, then the DDC algorithm with a window size of 23 is strictly superior to any static algorithm.

The rest of the paper is organized as follows. In the next section, we give a precise definition of our model. In Section 3, we make a detailed mathematical analysis of the Static Divergence Caching algorithm; particularly, we determine the optimal refresh rate for the case where the probability distribution of the random process generating the read and write requests is known and fixed over time. Section 4 presents the Dynamic Divergence Caching algorithm, which is appropriate when the probability distribution of the random process changes over time or is simply unknown. Section 5 gives a theoretical analysis of that algorithm's performance. In Section 6 we give experimental results obtained by simulating our techniques on various data sets. We compare this paper to the relevant literature in Section 7, and we make a few concluding remarks in Section 8.

2 Model of Problem

Our client-server system consists of a server computer, SC, and a client computer, CC. We consider a single data object that is always updated and stored at the SC, and that the CC requests a copy of it from time to time. In this paper we concentrate on the case where the CC may not need the absolutely most recent version of the data object.

A *request* is either a write, denoted w, issued by the SC or a read issued by the CC.[1] Each write request creates a new version of the object. Each read request has a *tolerance* t, specifying how recent a version of the object is required. We denote such a read by $r(t)$. We assume that the read tolerance is an integer in the range $1, \ldots, M$. A read tolerance of 1 indicates that the most recent version of the object is requested; a read tolerance of k indicates that any of the last k versions will do.

A *schedule* is a finite sequence of requests; for example, $w, w, r(1), w, r(3), r(2), w$. For the purpose of analysis,

[1] In general, there may also be writes issued by the CC and reads issued by the SC. However, since the SC always has the latest copy of the object, those requests always have the same fixed cost and therefore we ignore them.

we assume that the requests are sequential. The reason for this is that the reads are all generated by the CC, so they are sequential. Although a read and a write request or two write requests could be generated concurrently, at some point a concurrency control mechanism will serialize them, so our analysis still holds.

Throughout, we assume that requests are generated by Poisson processes. In particular, we assume that there are M separate Poisson processes, each generating read requests with a tolerance $1 \leq i \leq M$; let process i have intensity λ_{r_i}. Thus λ_{r_i} is the average number of read requests with tolerance i per time unit. Another Poisson process generates writes at the SC; call its intensity λ_w.

Another possible model of our problem would involve a single Poisson process generating read requests, with the tolerance of a read being determined by picking from the set of tolerances $\{1, 2, \ldots, M\}$ according to some fixed probability distribution. This model, however, is equivalent the model we specified above, since the sum or combination of M Poisson processes is itself a Poisson process.

One of the problems studied in this paper is, for the Static Divergence Caching algorithm, how often the CC should have the object regularly delivered. We will call this parameter the *refresh rate*. A refresh rate of k means that the object is automatically transmitted to the CC every time the object has been updated by k writes without having been sent to the CC in the meantime. Thus, when the refresh rate is k, the CC always has one of the k most recent versions of the object in its local memory, and can satisfy any read request with a tolerance of k or more with that local version. The CC may choose never to have the object regularly delivered; this corresponds to a refresh rate of infinity.

Observe that a refresh rate of k does not mean that exactly one in every k writes is propagated to the CC. If the CC solicits a refresh (to satisfy a read with a low tolerance) after $k/2$ writes, then that refresh will reinitialize the refresh counter. In other words, an automatic, or solicited refresh will occur after k writes only if in the meantime there was no solicited refresh. Observe that performing an unsolicited refresh exactly once for every k writes would strictly increase the number of object transfers compared to our proposed scheme.

We assume that each message containing the data object costs 1, and each read request, i.e. a control message, costs ω. Intuitively, $0 \leq \omega < 1$, but we will not assume this unless we explicitly say so. In our model, reads with tolerance less than the refresh rate cost $1 + \omega$, because the CC must send a control message of cost ω to "special order" the data object, and then pay 1 for the transmission of the data object. Reads with tolerance greater than or equal to the refresh rate have zero cost. A write costs 1 if it is propagated to the CC, otherwise it costs 0.

Note that the case $\omega = 0$ is of particular interest. It models the situation in mobile computing where communication is by cellular telephone calls, where there is a charge for the first minute or part thereof. If we assume that a remote read request and the response are executed within one minute, then each remote read or propagated write costs 1.

3 Fixed and known distributions

In this section we assume that the λs and ω are fixed and known a priori, and we are using the Static Divergence Caching algorithm. We develop the expected cost per time unit as a function of the refresh rate, and we show how to find the minimum of that function. This minimum is the optimal refresh rate.

Our first goal is to compute the expected cost per time unit for any given fixed refresh rate k. (The time unit is the period of time for the intensities λ; i.e. λ_w is the expected number of writes during one time unit). The case $k = \infty$ is straightforward. We pay $1+\omega$ for every read, and nothing for any of the writes. Thus the cost per unit time is $(1+\omega)\sum_{t=1}^{M} \lambda_{r_t}$. The case $k = 1$ is also straightforward. We pay 1 for each write and nothing for any reads, so the expected cost per unit time is λ_w.

Otherwise, for fixed integer $1 < k < \infty$, the requests we might possibly pay for are writes, and reads with tolerance less than k. We call such requests *relevant* and reads with tolerance at least k *irrelevant*. Notice that irrelevant requests are always free. Notice also that, as explained in the previous section, the number of writes we pay for is not $1/k$'th of all the writes (which would make the derivation of the optimal refresh rate easier).

Put $r(k) = \sum_{t=1}^{k-1} \lambda_{r_t}$. This is the intensity of the Poisson process generating all reads that we will have to pay for; once we have fixed k the individual values of the λ_{r_t} for $t < k$ no longer matter.

Put $\theta_k = \lambda_w/(\lambda_w + r(k))$, and put $\theta = \theta_{M+1}$. (Notice that $\theta_1 = 1$.) Thus θ is the probability that an arbitrary request is a write, and the probability that a relevant request is a write is θ_k.

Now from here on in, we condition all probabilities and expectations on the event that the request being considered is relevant. (At the end, we will need to multiply through by the probability of this event, which is $(\lambda_w + r(k))/(\lambda_w + r(k) + \sum_{i=t}^{M} \lambda_{r_i})$.)

The probability that we pay for an arbitrary request is the sum of the probabilities that the request is a relevant read, which is clearly $1 - \theta_k$, and that the request is a write we pay for. Hence we need to calculate the probability that a relevant request is a write we pay for. In fact, we pay for a write in a schedule whenever the sequence of relevant requests leading up to and including that request is either a read followed by k writes, or a read followed by $2k$ writes, or a read followed by $3k$ writes, or.... These events are all disjoint, so the probability of any of them occurring is just the sum of the probability of each. This sum in the limit is

$$\Pr\left[\text{Request is write we pay for}\right] = \sum_{n=1}^{\infty}(1-\theta_k)(\theta_k)^{nk} = \frac{(1-\theta_k)\theta_k^k}{1-\theta_k^k}.$$

Notice that the precise probability depends on how long the communications have been taking place. However, the geometric sum converges very quickly, so the limiting value calculated above should be a very close approximation after the first few requests.

Thus the expected cost of one arbitrary relevant request is

$$E(\text{cost}) = (1-\theta_k)\left(1+\omega+\frac{\theta_k^k}{1-\theta_k^k}\right). \quad (1)$$

Recall that this was conditioned on the request being relevant, so the actual expected cost of one arbitrary request is

$$\frac{(r(k)+\lambda_w)}{\lambda_w + r(k) + \sum_{i=k}^{M}\lambda_{r_i}}(1-\theta_k)\left(\omega+\frac{1}{1-\theta_k^k}\right) = r(k)\left(\omega+\frac{1}{1-\theta_k^k}\right) \Big/ \left(\lambda_w + r(k) + \sum_{i=k}^{M}\lambda_{r_i}\right) \quad (2)$$

since, by the definition of θ_k, $(r(k)+\lambda_w)(1-\theta_k) = r(k)$.

The expected cost per unit time is the quantity specified by Equation 2 times the expected number of requests per unit time, which is just the denominator of that fraction. Thus for a refresh rate of $1 < k < \infty$ the expected cost per unit time is

$$\text{E(cost)} = r(k)\left(\omega+\frac{1}{1-\theta_k^k}\right). \quad (3)$$

Intuitively, Equation 3 reflects the balancing of two opposite influences on the cost. As k increases, $r(k)$ increases. That is, we pay more for "special orders" if we have a higher refresh rate. On the other hand, as k increases, the factor $1/(1-\theta_k^k)$ decreases, since for a fixed value of θ_k it would decrease as the exponent k increases, and in addition, θ_k decreases with k. This factor corresponds to the fact that as k increases the amount we pay for writes at the SC decreases.

Putting Equation 3 together with the extreme values for the refresh rate we get that the expected cost per unit time for a fixed refresh rate k is

$$\text{E(cost)} = \begin{cases} \lambda_w & \text{for } k=1 \\ \left(\omega+\frac{1}{1-\theta_k^k}\right)\sum_{j=1}^{k-1}\lambda_{r_j} & \text{for } 1<k<\infty \\ (\omega+1)\sum_{j=1}^{M}\lambda_{r_j} & \text{for } k=\infty \end{cases} \quad (4)$$

Theorem 1 *In the Static Divergence Caching algorithm the minimum cost per unit time can never be achieved for any finite refresh rate greater than M.*

Proof Straightforward based on the cost function in Equation 4 that for any $k > M$ the second line is bigger than the third line. □

Thus, assuming that all the λ's are fixed and known, the algorithm for finding the optimal refresh rate is trivial. All one must do is compute the $M+1$ different costs associated with the refresh rates of $1, 2, \ldots, M$ and ∞, according to Equation 4, and choose the minimum cost refresh rate.

4 The dynamic divergence caching algorithm

The Dynamic Divergence Caching algorithm works for distribution parameters (λ's) that are unknown and that may vary over time. The algorithm varies the refresh rate of an object x at the CC. It does so by computing the λ's based on a window of the k latest relevant read and write requests, using formula 4 to recompute the optimal refresh rate, and establishing it as the new refresh rate. The new refresh rate may be different than the previous one since the λ's change in a sliding window.

Now we explain the algorithm in detail. Recall, the relevant reads are issued at the CC, and the relevant writes are issued at the SC. At any point in time there is a refresh rate, r. Each read at the CC with a tolerance higher than r is satisfied locally, and each read at the CC with a tolerance lower than r results in a refresh request to be sent to the SC; the SC responds by refreshing x, i.e., sending the latest version of x. The SC also performs an unsolicited refresh of x when it receives r consecutive write requests since the last refresh of x (this last refresh may be either solicited or unsolicited).

Adaptation of the refresh rate occurs at each refresh point (solicited or not), as follows. At every point in time, the SC maintains the *write-sliding-window*, i.e. the set of times of the last k write requests. Each time a new write is received at the SC its time stamp is added to the write-sliding-window, and the smallest time stamp in the window is deleted. At every point in time, the CC maintains the *read-sliding-window*, i.e. the time and tolerance of the k

latest reads. Specifically, the CC maintains a set of k pairs (i, t) where i is a tolerance and t is a time stamp.

When the CC initiates a refresh request, the SC computes the new refresh rate as follows. The CC piggybacks the read-sliding-window on each refresh request. Before refreshing x, the SC uses the times in the read-sliding-window and the write-sliding window in order to compute the *request-numbers-window*; it is the number of writes, the number of reads with tolerance 0, the number of reads with tolerance 1, the number of reads with tolerance 2, etc. for the last k read-write requests. These numbers are taken as the λ_w and λ_r's. Then the SC uses Formula 4, in order to compute the optimal refresh rate; this will become the new refresh rate. Then the SC responds to the refresh request, and it informs the CC of the new refresh rate by piggybacking the rate on the refresh value sent to the CC.

When the SC performs an unsolicited refresh, the CC computes a new refresh rate as follows. The SC piggybacks the *write-sliding-window* on the unsolicited refresh. The CC uses these times to compute the request-numbers-window for the last k read-write requests. These numbers are taken as the λ_w and λ_r's, and the CC uses Formula 4 in order to compute the optimal refresh rate; this will become the new refresh rate, and the CC has to send a control message to the SC informing it of the new refresh rate. Thus, there is an extra cost incurred if the CC changes the refresh rate. Thus the CC will change the refresh rate only if the expected cost for the optimal refresh rate beats the cost of the current refresh rate by at least ω.

Before concluding this section, we make two remarks about the algorithm. First, it may be necessary to take into consideration that the CC and SC clocks are not synchronized. This may be a problem since the algorithm computes the request-numbers-window by selecting, from the read and write sliding windows, the *latest* k requests. In order for this set to be correctly computed, one has to account for the clocks' divergence. But this can be done easily if, once every g refreshes, the SC piggybacks the current value of its clock on the refresh message. Then the CC can compute the clocks' divergence, and either synchronize its clock to the SC's clock, or adjust the times in the sliding window to account for the divergence.

The second remark concerns k, the size of the sliding window. There are two opposite forces to be considered in choosing the value of k. If the values of the λ's change over time, then choosing a smaller k has the effect of discarding past λ's in favor of recent ones, i.e. adjusting to the current λ's. On the other hand, assume that each type of request is Poisson distributed with a fixed parameter. Then, by the law of large numbers, the larger the value of k, the closer are the values in the request-numbers-window to the actual λ's. However, in [6] we have shown that, even if the distribution parameters are fixed (but unknown), for a window of size 23, the expected cost of a sliding window algorithm comes within 4% of the optimum expected cost.

Thus we feel that a window of approximately 23 is appropriate, and using it the DDC algorithm will perform well even if the distribution parameters are fixed.

5 Worst case

In this section we analyze the worst case behavior of the Dynamic Divergence Caching algorithm and the Static Divergence Caching algorithm. We prove that the DDC algorithm is superior to the SDC algorithm.

The appropriate measure of the worst-case behavior of on-line algorithms is its *competitiveness* [10]. An on-line algorithm is an algorithm that receives its input schedule one request at a time, and acts on each request before obtaining the next one. Roughly speaking, an on-line algorithm is competitive if its cost is at worst a constant times the cost of any off-line algorithm, for any sequence of requests.

What is wrong with the traditional worst-case behavior of an algorithm? The answer is that the worst case of an on-line algorithm occurs when the input is chosen by a potent adversary trying to make the algorithm perform poorly. Then it makes no sense to ask "For Algorithm X, what is the maximum cost per request of any schedule?" The reason is that the adversary can construct a schedule by always making the next request one the algorithm must pay for. For example, let the control-message cost $\omega = 0$. For every algorithm considered in this paper, there is some schedule of length ℓ on which that algorithm incurs cost ℓ, so by that "worst-case" measure, all the algorithms have the same complexity.

What we really want to know is "How does the cost incurred by Algorithm X compare to the minimum cost that any algorithm must incur?" and it is this question that competitiveness answers.

Now let us give a precise definition of competitiveness. For an algorithm A and a schedule of requests σ, let $C_A(\sigma)$ denote the cost of Algorithm A on schedule σ. Then, on-line algorithm A is c-competitive if there is a constant k such that: for *any* schedule σ and *any* algorithm B (including all off-line algorithms that receive the entire schedule σ in advance) we have

$$c_A(\sigma) \leq c \cdot C_B(\sigma) + k.$$

An algorithm is *competitive* if it is c-competitive for some constant c.

For our problem, if requests really are generated by Poisson processes with fixed intensities, then one should use the algorithm given in Section 3 above, and competitiveness is an inappropriate criterion. However, if the requests are generated at random, in a way that makes the past requests irrelevant in predicting the future ones, static algorithms can perform very poorly.

In this section we will show that the DDC algorithm of Section 4 has a bounded competitive ratio. We begin by presenting the optimal off-line algorithm, that we call O, for an arbitrary schedule.

Let σ be an arbitrary schedule. We break σ into blocks of reads and writes. Formally let $B_1, B_2, \ldots, B_\ell$ be the division of σ into blocks such that each block is either all reads or all writes, and adjacent blocks contain different kinds of requests. Assume that at the beginning of σ the CC has a fresh version of the data object.

Off-line algorithm O marks read blocks as follows. The first read block that is marked is the first read block preceded by a greater total number of writes than the minimum tolerance of any read in the block. Thereafter, each time there is a read block containing a read with tolerance less than the total number of writes since the last write paid for, Algorithm O marks that read block. Algorithm O propagates to the CC the last write in every write block preceding a marked read block. All the other reads and writes are local, and consequently incur 0 cost. Thus, the cost of O is the total number of marked read blocks.

Lemma 1 *Algorithm O is the optimal off-line algorithm.*[2]

Theorem 2 *The Static Divergence Caching algorithm has an unbounded competitive ratio.*

On the other hand, the DDC algorithm performs well, in the sense of being competitive. To prove this, we first need to examine just how well the optimal off-line algorithm performs.

Lemma 2 *Between any two read blocks marked by Algorithm O, there are at most* $2M$ *blocks.*

Theorem 3 *The Dynamic Divergence Caching algorithm is competitive.*

6 Experimental results

We have performed many experiments, where we randomly generated schedules of requests, and compared the algorithms' performance on those schedules. In this section, we summarize the results. In both subsections we compare the Dynamic Divergence Caching algorithm (hereinafter DDC) to the the Static Divergence Caching algorithm (hereinafter SDC). The difference between the subsections lies in the selection of the input schedules. In the first subsection the input schedules have fixed distribution parameters (λ's), and in the second subsection they vary over time. For all our experiments, we fixed the maximum tolerance of any read, denoted M, at 20.

6.1 Fixed distribution parameters

We generated schedules of 1600 requests, for each of 84 different values of the 21 parameters λ_w and λ_{r_j} for $1 \leq j \leq 20$. For each schedule σ we used formula 4 to compute the optimal refresh rate, k_σ. Then we ran on σ the DDC algorithm with various window sizes, and the SDC(k_σ) algorithm.[3]

In Figure 1 we show a grand summary of the data — the average of all 84 runs, for the case where the control message cost is negligible, namely $\omega = 0$. In particular, we plot the ratio of the average (over all runs) cost of the DDC algorithm with window size k (hereinafter DDC(k)) to the average cost of the optimal static algorithm, as a function of k. Notice that the optimal static algorithm differs from schedule to schedule.

The solid line at 1 represents the average cost of the optimal static algorithm for each case.

The main result of these experiments is that the performance of $DDC(k)$ improves sharply as k increases to about 23, and for $k \geq 23$ the cost of DDC(k) is only 10–15% greater than the cost of the best static algorithm. In fact, this pattern was observed for almost every schedule of the experiment, although the graph in Figure 1 shows only data for all the runs averaged together.

Two other important quantities are *not* shown in the graph. One is the performance of the optimal off-line algorithm. The average cost of the optimal static algorithm was typically about 2.25 times the average cost of the optimal off-line algorithm. Recall that the off-line algorithm has the advantage of seeing all the requests in advance, something that is impossible in real life.

We also compared the optimal static algorithm to the better (on the particular run) of the static algorithms with refresh rates 1 and infinity. This corresponds to traditional methods that do not allow for divergence caching, but only for caching (refresh rate 1) or not caching (refresh rate infinity) [6]. The optimal static algorithm showed, on average, a factor of two improvement. This demonstrates the power of divergence caching.

Note that the factor of two improvement is an average over 84 different settings of the λs. For certain values of the λs a refresh rate of 1 or infinity was the optimal value (typically with writes being either a very high or very low percentage of all requests), and for certain values of the

[2] Proofs are omitted because of space limitations.

[3] SDC(t) is the SDC algorithm with refresh rate t.

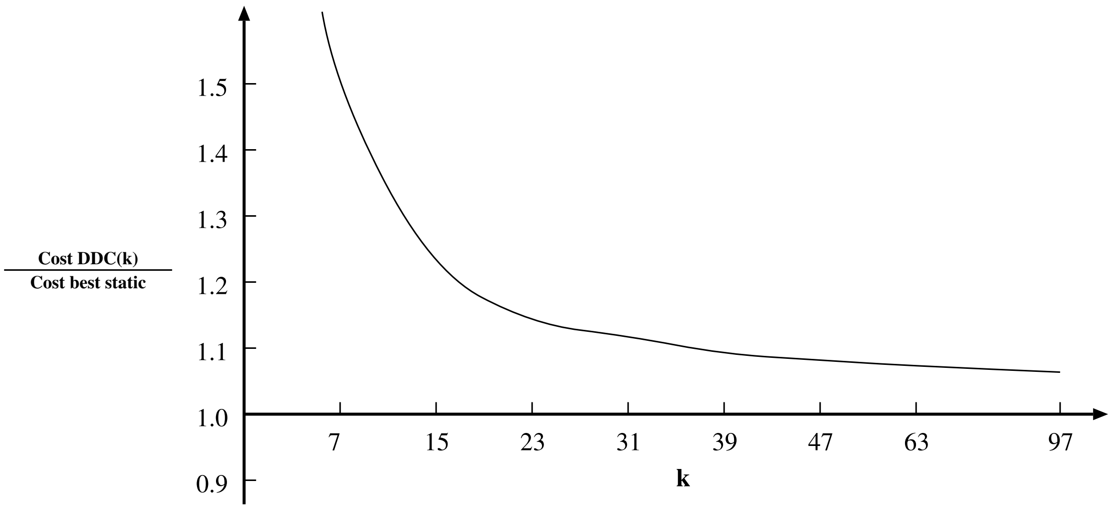

Figure 1: Experimental results for fixed λ's for $\omega = 0$ case. Graph shows ratio of costs paid by DDC(k) to costs paid by the optimal static algorithm.

λs the improvement with divergence caching was much greater than fourfold.

We also computed the performance of our algorithms for values of ω ranging from 0.1 to 0.9. We performed the same experiments as for the case of $\omega = 0$. The results were broadly similar, with the following exception.

DDC(k)'s performance became worse as ω increased, especially for small values of k. For example, for the combination of experiments reported in Figure 1, with $\omega = 0$, DDC(23) paid 14.9% more than the optimal static divergence algorithm. However, for a large value of ω, it was clearly significantly better to use the best static algorithm than any dynamic algorithm. For $\omega = 0.9$, DDC(23) paid 44.1% more on average than the best static algorithm. We hypothesize that the high costs for the dynamic algorithms are caused by the cost of the control messages to reset the refresh rate.

6.2 Time-varying distribution parameters

As we describe above in Section 5, theory leads us to believe that if the λ's change over time, then the dynamic divergence caching algorithms will outperform all static algorithms. We ran several experiments where we varied the λs over time, and we summarize the results here. They confirm our expectation.

We begin with the case of $\omega = 0$. A typical experiment is presented in Figure 2, which reports the average of 40 runs of 1600 requests each. In each run, we picked each of the λ_{r_j}s uniformly at random from 1 to 100, and then uniformly at random picked the fraction p of all requests that would be writes. (Thus λ_w was set to be $p/(1-p)$ times the sum of the λ_r's.) Every 157 requests, the λ's were randomly assigned new values according to the same rules.

We determined empirically which one refresh rate gave the best performance of all the static algorithms over the forty runs. In figure 2 we plot the ratio of the average (over all 40 runs) cost of DDC(k) to the average cost of that static algorithm. The main result is that the cost of DDC(k) improves with k, with considerable improvement up to around $k = 23$, and slight improvement thereafter. For $k \geq 23$, the average cost of DDC(k) is roughly 70% of the cost of the best static algorithm.

The cost-advantage of the dynamic divergence caching algorithms varied with the method of changing the λs over time. If the ratio of writes to reads remained constant over time, while the individual λ_{r_j}s varied, then the dynamic divergence algorithms were only slightly better than the best static algorithm. On the other hand, much larger improvements than those shown in Figure 2 were found when we alternated periods where writes were at most 30% of all requests with periods where writes were at least 70% of all requests. In practice, alternating phases of read-intensive and write-intensive patterns might be a common.

Remember that these results are for the case $\omega = 0$. We ran the same experiments for values of ω ranging from 0.1 to 0.9. We once again found that the dynamic divergence algorithms DDC(k) are superior to the best static algorithm, again with a value of $k = 23$ seeming to be about where the gain levels off. However, the cost-advantage of DDC(k) decreased as ω increased. For example, for $\omega = 0.1$ the advantage of DDC(23) was 19.1% (versus an advantage of 25.6% for $\omega = 0$). The cost-advantage decreased roughly

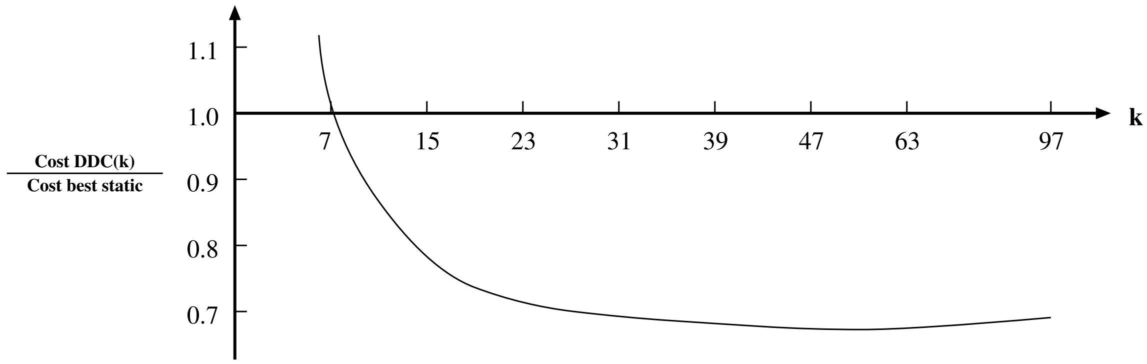

Figure 2: Experimental results for time-varying λ's for $\omega = 0$ case. Graph shows ratio of costs paid by DDC(k) to costs paid by the optimal static algorithm.

linearly with ω to become near zero at $\omega = 0.9$. This is due to the increasing cost of control messages to reset the CC's refresh rate.

7 Relevant research

This paper is related to two topics, caching and replica divergence. We will first discuss related works on caching. Such works (e.g. [5, 11, 10, 13]) have assumed a consistent environment, i.e. one in which cached objects are copies of the most recent version. In our previous works (e.g. [6, 7, 8]) we have also made this assumption, and we have concentrated on a variant of caching called dynamic allocation. The objective of dynamic allocation is to minimize communication, i.e. the number of object transfers, rather than optimizing performance as in traditional caching. For example, given enough storage, traditional caching will keep a copy of every accessed object at the client computer; if the number of writes at the server computer is much higher than the number of reads at the client computer, then this approach will incur excessive communication. In contrast, dynamic allocation caches and uncaches an object depending on the read-write pattern.

Dynamic allocation in a weak-consistency environment, as studied in this paper, is a generalization of dynamic allocation in a strong consistency environment. Specifically, in a strong consistency environment a client computer is in one of two states. If the client computer has a copy of the data object, it means that the refresh-rate is 1 (i.e. every update is propagated to the client computer). If it does not have a copy of the object, then the refresh-rate is infinity. Dynamic allocation switches the refresh rate of a client computer between two values, 1 and ∞.

In a weak consistency environment these two states are extremes of a spectrum. Dynamic allocation permits intermediate values of the refresh-rate. Furthermore, the refresh-rate may vary dynamically depending on the read-write pattern. In other words, in a weak consistency environment the access pattern of an object determines not only the allocation scheme, but also the frequency at which the object is refreshed.

Now we will discuss related works on replica divergence. There are several studies of replica-divergence issues, such as [1, 2, 4, 12, 14]. These works assume, as we do, that each object has a computer that stores the most up to date version, and other computers store quasi-replicas that may diverge. The allowed divergence is specified by the user. In other words, a user manually specifies for each quasi-replica of an object a fixed divergence which results in a fixed refresh rate. For example, in [2], if the quasi-replica of object x at a particular computer c has a refresh rate of three, then every third update generates a refresh of x at c. Therefore, the divergence is specified at the quasi-replica level.

In contrast, in this paper we assume that the user specifies the allowed divergence, i.e. the tolerance, at the read level. The divergence of the quasi-replica is computed by an algorithm that we provide. It automatically varies the quasi-replica divergence, i.e. the refresh rate, in order to optimize costs. This optimization is not guaranteed by previous works. Moreover, the previous works do not discuss how to determine the refresh rate. The formulas that we obtain in this paper can be used for this purpose (assuming that the distributions of the reads with the various tolerances are known), and in this sense the present work is complementary to previous work.

8 Concluding remarks

In this paper we proposed a new mechanism, divergence caching, for use in client-server computing environments to reduce the number of transmissions from database servers to clients. Divergence caching achieves this objective by giving a tolerance to each read of an object issued at a client computer, namely an allowed deviation from the most up-to-date version of the object; this is combined with a periodic refresh, namely a periodic transmission of the object from the on-line database to the client.

We proposed two algorithms based on divergence caching, Static and Dynamic. In static divergence caching the refresh rate is fixed, whereas in dynamic it varies over time. The first is appropriate when the access pattern to an object is fixed and known a priori, and the latter is appropriate in other cases.

These algorithms were analyzed in the worst case and in the expected case. We have shown that in the worst case the dynamic algorithm is strictly superior to the static one.

The results of the probabilistic analysis are as follows. We obtained a formula (4) for the optimal refresh rate of the static divergence caching algorithm, i.e. the refresh rate that minimizes the expected cost of the algorithm. The optimal refresh rate depends on the distributions of reads and writes of the object, and on the ratio of control-message-cost to data-message-cost.

Then we experimentally analyzed the dynamic algorithm using a large number of schedules that were probabilistically generated using Poisson processes. An important parameter of the dynamic algorithm is the size of the sliding window. We have shown that the appropriate window size is about 23.

We also showed that when the distributions of reads and writes are fixed, and a control-message-cost is relatively small compared to a data-message-cost, then the dynamic algorithm comes within 15% of the optimal static one; when the cost of the control message is higher the dynamic algorithm performs worse. The reason for this is that the dynamic algorithm sends additional control messages for the client computer to inform the server of refresh rate changes.

When the distributions of reads and writes vary over time, then the dynamic algorithm is superior to any static one, regardless of the ratio of control-message-cost and data-message-cost.

References

[1] R. Alonso, D. Barbara, and H. Garcia-Molina. Quasi-copies: Efficient data sharing for information retrieval systems. In *Proc. of EDBT '88, LNCS 303*. Springer Verlag, 1988.

[2] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Trans. Database Syst.*, 15(3):359–384, 1990.

[3] B. R. Badrinath and T. Imielinski. Replication and mobility. In *Proceedings of the Second Workshop on the Management of Replicated Data*, pages 9–12.

[4] D. Barbara and H. Garcia-Molina. The case for controlled inconsistency in replicated data. In *Proc. of the IEEE Workshop on replicated data*, 1990.

[5] M. J. Carey, M. J. Franklin, M. Livny, and E. J. Shekita. Data caching tradeoffs in client-server DBMS architectures. In *Proc. 1991 ACM SIGMOD Internat. Conf. on Management of Data*, pages 357–366, 1991.

[6] Y. Huang, P. Sistla, and O. Wolfson. Data replication for mobile computers. In *Proc. 1994 ACM SIGMOD Internat. Conf. on Management of Data*, pages 13–24.

[7] Y. Huang and O. Wolfson. A competitive dynamic data replication algorithm. In *Proc. Ninth Internat. Conf. on Data Engineering*, pages 310–317, Vienna, Austria, 1993. IEEE.

[8] Y. Huang and O. Wolfson. Dynamic allocation in distributed systems and mobile computers. In *Proc. Tenth Internat. Conf. on Data Engineering*, pages 20–29, Houston, Texas, 1994. IEEE.

[9] T. Imielinski and B. R. Badrinath. Querying in highly mobile distributed environments. In *Proc. of the 18th International Conference on VLDB*, pages 41–52, 1992.

[10] A. Karlin, M. Manasse, L. Rudolph, and D. Sleator. Competitive snoopy caching. *Algorithmica*, 3:79–119, 1988.

[11] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[12] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A snapshot differential refresh algorithm. In *Proc. ACM SIGMOD Internat. Conf. on Management of Data*, pages 53–60, 1986.

[13] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Softw. Eng.*, 39(4):447–459, Apr. 1990.

[14] D. B. Terry. Caching hints in distributed systems. *IEEE Trans. Softw. Eng.*, pages 48–54, Jan. 1987.